



BIGDATA DATABASE SYSTEM

Jitendra Kumar¹, Atteeq Ahmed²

¹Assistant Professor, ²Associate Professor,
IIMT College of Management, Greater Noida, (India)

ABSTRACT

The HDFS plays a role to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications in Bigdata. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size.

Keywords: Hadoop, HDFS, Distributed File System

ARCHITECTURE OF BIGDATA DATABASE SYSTEM

A. NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by *inodes*, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes (the physical location of file data). An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

B. DataNodes

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's *generation stamp*. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive. During startup each DataNode connects to the NameNode and performs a *handshake*. The purpose of the handshake is to verify the *namespace ID* and the *software version* of the DataNode. If either does not match that of the NameNode the DataNode automatically



shuts down. The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus preserving the integrity of the file system. The consistency of software versions is important because incompatible version may cause data corruption or loss, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the DataNode *registers* with the NameNode. DataNodes persistently store their unique *storage IDs*. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that.

A DataNode identifies block replicas in its possession to the NameNode by sending a *block report*. A block report contains the *block id*, the *generation stamp* and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster.

During normal operation DataNodes send *heartbeats* to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions.

C. HDFS Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface. Similar to most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas. When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas



of the first block of the file. The client organizes a pipeline from node-to-node and sends the data.

When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different.

Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth.

D. Image and Journal

The namespace image is the file system metadata that describes the organization of application data as directories and files. A persistent record of the image written to disk is called a *checkpoint*. The journal is a write-ahead commit log for changes to the file system that must be persistent. For each client-initiated transaction, the change is recorded in the journal, and the journal file is flushed and synched before the change is committed to the HDFS client. The checkpoint file is never changed by the NameNode; it is replaced in its entirety when a new checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next section. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal until the image is up-to-date with the last state of the file system. A new checkpoint and empty journal are written back to the storage directories before the NameNode starts serving clients.

If either the checkpoint or the journal is missing, or becomes corrupt, the namespace information will be lost partly or entirely. In order to preserve this critical information HDFS can be configured to store the checkpoint and journal in multiple storage directories. Recommended practice is to place the directories on different volumes, and for one storage directory to be on a remote NFS server. The first choice prevents loss from single volume failures, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available.

E. CheckpointNode

The NameNode in HDFS, in addition to its primary role serving client requests, can alternatively execute either of two other roles, either a *CheckpointNode* or a *BackupNode*. The role is specified at the node startup. The



CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The CheckpointNode usually runs on a different host from the NameNode since it has the same memory requirements as the NameNode. It downloads the current checkpoint and journal files from the NameNode, merges them locally, and returns the new checkpoint back to the NameNode.

Creating periodic checkpoints is one way to protect the file system metadata. The system can start from the most recent checkpoint if all other persistent copies of the namespace image or journal are unavailable.

Creating a checkpoint lets the NameNode truncate the tail of the journal when the new checkpoint is uploaded to the NameNode. HDFS clusters run for prolonged periods of time without restarts during which the journal constantly grows. If the journal grows very large, the probability of loss or corruption of the journal file increases. Also, a very large journal extends the time required to restart the NameNode. For a large cluster, it takes an hour to process a week-long journal. Good practice is to create a daily checkpoint.

F. BackupNode

A recently introduced feature of HDFS is the *BackupNode*. Like a CheckpointNode, the BackupNode is capable of creating periodic checkpoints, but in addition it maintains an inmemory, up-to-date image of the file system namespace that is always synchronized with the state of the NameNode. The BackupNode accepts the journal stream of namespace transactions from the active NameNode, saves them to its own storage directories, and applies these transactions to its own namespace image in memory. The NameNode treats the BackupNode as a journal store the same as it treats journal files in its storage directories. If the NameNode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state. The BackupNode can create a checkpoint without downloading checkpoint and journal files from the active NameNode, since it already has an up-to-date namespace image in its memory. This makes the checkpoint process on the BackupNode more efficient as it only needs to save the namespace into its local storage directories.

G. Upgrades, File System Snapshots

During software upgrades the possibility of corrupting the system due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades. The snapshot mechanism lets administrators persistently save the current state of the file system, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot.

The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint



and journal remain unchanged. During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the data files directories as this will require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories.

The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot, thus finalizing the software upgrade.

System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The *layout version* identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version.

HDFS does not separate layout versions for the NameNode and DataNodes because snapshot creation must be an allcluster effort rather than a node-selective event. If an upgraded NameNode due to a software bug purges its image then backing up only the namespace state still results in total data loss, as the NameNode will not recognize the blocks reported by DataNodes, and will order their deletion. Rolling back in this case will recover the metadata, but the data itself will be lost. A coordinated snapshot is required to avoid a cataclysmic destruction.

REFERENCES

- [1] S. Ghemawat, H. Gobiuff, S. Leung. "The Google file system," In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29–43.
- [2] F. P. Junqueira, B. C. Reed. "The life and times of a zookeeper," In Proc. of the 28th ACM Symposium on Principles of Distributed Computing, Calgary, AB, Canada, August 10–12, 2009.
- [3] Lustre File System. <http://www.lustre.org>
- [4] M. K. McKusick, S. Quinlan. "GFS: Evolution on Fast-forward," ACM Queue, vol. 7, no. 7, New York, NY. August 2009.

- [5] O. O'Malley, A. C. Murthy. Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds. May 2009.
- [6] R. Pike, D. Presotto, K. Thompson, H. Trickey, P. Winterbottom, "Use of Name Spaces in Plan9," *Operating Systems Review*, 27(2), April1993, pages 72–76.
- [7] S. Radia, "Naming Policies in the spring system," In Proc. of 1st IEEE Workshop on Services in Distributed and Networked Environments, June 1994, pp. 164–171.